Software Product-Lines: What To Do When Enumeration Won't Work

David Lorge Parnas
Software Quality Research Laboratory (SQRL)
Department of Computer Science and Information Systems
Faculty of Informatics and Electronics
University of Limerick, Limerick, Ireland
David.Parnas@ul.ie

Abstract

The history of research on the development of program-families is briefly reviewed.

Two distinct problems, configuration-management *and* family-design *are identified. It is explained that, while software configuration-management is not fundamentally different from configuration-management for other products, in practice, inadequate attention to family-design exacerbates all problems associated with developing and maintaining program families. It is suggested that although enumeration is useable for configuration-management, product-line design by enumeration is not generally feasible.*

An alternative approach, family member characterization using abstract documentation, is discussed. This approach is practical for family-design and can make configuration-management easier.

The advantages of designing an interface in terms of programs over an interface expressed as a data structure using conventions such as XML are also discussed.

1. Designing program-families: early history

I first observed an industrial effort to develop a family of programs, in this case a product-line, in 1969. The developer's plan was to develop three operating systems independently and, *after all three were completed and functioning*, meet to make them compatible for users and software. Even the limited software development experience accumulated by that time, made it clear that this was not likely to succeed. The earliest design decisions in a project are usually the most difficult to revise. If we want programs to implement a common interface, it is important to agree on that interface at the start of development rather than to try to revise the programs after completion[1].

These thoughts were generalized and published in [PA76], which was, as far as I know, the earliest paper to discuss the design and development of program-families. In this paper, I defined "*program-family*" by stating that, "*A set of programs constitutes a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members*"[2]. The essence of that paper was simple - when developing a family of programs, develop, **and document**, the common/shared aspects first. The *documented decisions* must be binding on the developers who work on the individual family members.

I had a second encounter with the problem of multi-version software when designing software for the U.S. Navy's A-7 aircraft [Pa77, NRL80, NRL81]. In this application, frequent replacement of peripheral devices was the rule rather than the exception and the software should have been designed in such a way that the software needed to operate new versions of these devices, could be written and "plugged in" quickly. In the original software, the substitution of a new device usually required difficult, and widespread, modification

[1] Of course one can hide that interface but this requires agreeing on another, more abstract, interface. Only certain aspects of the interface are actually hidden.

[2] It should be noted that this definition encompasses sets of programs that would not be included in what is commonly understood as a product-line. A program-family may include programs that are used for entirely different purposes.

of the code[3]. In our approach, the detailed characteristics of each of the devices would be "hidden" by a software device interface module that would be revised or replaced whenever a new type of device was installed[4]. The key to easing this replacement was to design and document an interface for the device dependent module that would not need to be changed even though the device, and the software that communicated directly with it had to be revised or replaced.

We viewed this problem as one of designing for families of such software modules. Again, we found that the common aspects of these modules, in this case the interfaces to the rest of the software could be developed before developing any members of the family. In other words, we had to design software for a program-family at a time when we could not possibly know what members of the family would actually be built. The papers cited above discuss principles and procedures for designing lasting interfaces before the hardware is designed, i.e. before a domain analysis could be done. It may seem paradoxical, but the key idea to designing for variation, was to find, **and document**, the commonalities. This will be discussed further below.

2. Program-families today

Today, more than 35 years later, many software-producing companies are faced with problems arising because they sell a large set of closely related products. For example, those who manufacture mobile phones must offer models that work in a variety of network connectivity protocols, offer a variety of features, and support a variety of provider specific services. As a result, I can buy two phones that look about the same but whose capabilities differ in many ways. The only difference may be the software.

Those who produce car radios, telephone switches, and desktop software are all faced with the same problems; all of their products share basic functions in common but the developers must deal with many different configurations of hardware, environmental interfaces, and user interfaces. The simultaneous maintenance of so many variants leads to problems that were not often encountered in the early days of software development. Because the interfaces of the software components are complex, there are combinations of "features[5]" that work badly or do not work at all. This leads to a different problem, one that is sometimes called *variability management*, but might better be called *configuration-management*; it is the problem of knowing which combinations will be useful and which make no sense.

It has often been observed that maintaining a set of almost alike products is more difficult than maintaining a set of radically different products. The similarities lead to confusion. The situation gets worse as each variant is maintained separately and they begin to diverge as each team fixes a problem in a different way.

Returning to my 30 year old definition, "*it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members*" we see that, because the commonalities are not documented, it is not easy to begin by studying them. As the products evolve, the commonalities often evaporate; the individuals maintaining individual family members often do not know that these properties should be preserved. In general, the commonalities that could or did exist are neither exploited nor maintained.

3. What is special about software product-lines?

I prefer the term "configuration-management" to "variability management" because the former term suggests that the problems are neither new nor unique to software. One finds the same problem with many types of products.

For example, in the airline industry, improvements and part-substitutions while a plane is in active service are commonplace. Moreover, because it is economically important to keep the aircraft in service, a part may be replaced by a version with a different internal structure and hence different components. The part that was removed can be repaired (with reduced time-pressure) and later installed on another aircraft. As a result of this "replace now, repair and reuse later" policy, a model type such as "A320" actually denotes a huge family of aircraft versions and the set of versions actually in service varies frequently. Further, the time-in-service of components in a given aircraft can be very different resulting in complex inspection and maintenance schedules. There has been at least one publicly known incident caused by substituting an "almost compatible" to get a plane "up" quickly. In

[3] This was considered quite typical. In fact, the original software was considered to be of high quality.

[4] The hardware did not allow us the luxury of having many alternative drivers in the software and having the software select the correct driver when a device is installed.

[5] an overused and under-defined term

other words, all of the complexity that software developers are experiencing in managing product-lines can found in aircraft fleet maintenance.

In spite of this, software researchers and developers seem to have the impression that software "variability management" is unique in its complexity. They have this impression for two, very different, reasons:

- Many software developers are unaware of the complexity of the configuration-management problem in other Engineering fields;
- It is especially difficult to "manage variability" in software because it is unreasonably difficult to predict which combinations of component versions or features will function properly. Whereas an automobile manufacturer is almost always aware that a certain engine variant may make a tyre type inappropriate, software developers are often surprised to discover that revising or replacing one application or component has implications for others. We have all experienced an "upgrade" that led to a problem with one of our other applications or components.

The cause of the unexpected difficulties that software purveyors experience when dealing with product-lines is easy to explain: the interfaces between software components are complex, poorly understood, and poorly documented. It is because of this that when one version of a component is substituted for another, there may be unexpected consequences. In other words, the problem of variability management is often exacerbated by the fact that the **design** was done badly[6].

4. Program-family design revisited

The problems that companies are experiencing when managing software product-lines illustrates a decades-old software design observation, *software projects are usually difficult to manage because the software is poorly designed*. In the software field, as in many others, we often see researchers and developers studying, ways to treat, the symptoms rather than trying to find and eliminate the cause of those symptoms. In the case of "product-line" development, we see two approaches that I consider to be instances of treating the symptoms rather than the disease. The symptoms that we researchers are trying to cure are:

- having a great many code artefacts including classes of components that are almost alike but not interchangeable, and
- having to deal with a large (but finite) set of functional and environmental alternatives some of which may be incompatible with others.

The "symptomatic relief" comes in the form of:

- emphasis on sharing **concrete** artefacts among family members, and
- attempting to design software product lines by enumerating alternatives discovered in a "domain analysis".

These will be discussed in more depth in below.

4.1. Commonalities need not be concrete code artefacts

Many current articles on software product-line development propose either the development of a set of common components (e.g. frameworks) or the development of tools that will be used to produce the members of the product-line. Reviewing the early definition of program-family given in [Pa76], it should be obvious that the commonalities need not be executable code; commonalities might include such things as a user's manual or even a "look and feel"; these can be shared by programs that have no instructions in common and were not generated by the same set of tools. There may be good reasons that make it impossible for a set of products to share any concrete artefacts, but we can still have them present a common interface to their user or share a common "architecture". If we want such abstract properties to be shared, we need to make the corresponding design decisions, **and document them**, early in the development process.

4.2. Enumeration is insufficient for designing program-families

The dilemma faced by those who want to make common decisions for product-lines is that, as discussed in [NRL80, NRL81] we rarely know all the variants that will be needed when we start to develop a program-family[7]; under those circumstances, it is impossible to describe the family by enumerating all the members[8]. However, it is well known that there are

[6] It can also be the case that the design was good and well-documented but later ignored. I have seen this but it is rare simply because good well-documented designs are rare.

[7] If we think we know, we are usually wrong.

[8] In my experience, if I point out a missing alternative to someone who has illustrated an enumerative approach to product-line development, they shrug their shoulders and remark that they cannot be expected to see into the future or to know everything. While that

two ways to describe sets, one can either enumerate the members or describe a characteristic predicate, a predicate on the set of possible members that is true if and only if applied to a member of the set. The design of program-families requires that we use characterization rather than enumeration. Enumeration may be appropriate when managing a fixed set of products, but it will not work when we try to prepare for future variants, i.e. when we plan ahead.

5. Documents as representations of the characteristic predicate.

As was mentioned above, software product lines are often difficult to manage because the component interfaces are not well understood or documented. One of the illnesses of the software development field is that interfaces are decided by implementers while programming, not by the architects or designers before implementation begins. Moreover, they are often defined by whoever "gets there first" with the result that they may not be optimal for others. Even when an interface is specified by the architects, the actually implementation often introduces undocumented communication channels or deviates in some undocumented but "necessary" way from what was specified.

Precise interface documents can solve two problems:

- they can make it possible to produce interchangeable versions of a component and thereby simplify the configuration-management problem,
- they can characterize a set of implementations even if some characteristics of some existing or future members of the set are not known.

Any implementation that conforms to the specified interface is a potential variant. Any program that can be written using the specified interface, and whose correctness could be proven using only information in the interface document, will work with any potential variant. When a new variant is introduced, the rest of the software is compatible with it provided that it satisfies the specification and the rest of the software assumes no additional properties.

As discussed in more detail in [Pa77, NRL81] the design of these interfaces is not a "quick-and dirty" job. It requires careful research to determine what is possibly a variant and, perhaps more important, what is **not**. Essentially, the designer must look for properties that all variants will possess and make sure that there are fundamental reasons for expecting those properties to be unchanged. These properties must relate to the functions that can be provided, not to the way that they are provided.

For example, an interface for a navigation device can be written so that it applies to rotatable gyroscopes, gyroscopes that are fixed in inertial space, laser-based inertial measurement systems, and GPS-based systems. Although these use very different implementations, they must all provide the same information; programs that provide that information can be specified as the interface.

[Pa77] and [NRL81] also discuss the design of more "powerful" interfaces that apply to a subfamily of the variants. When needed, we can define sub-families of the family, comprising products with additional properties, i.e. properties that are not shared by all members of the full family.

Of course the interfaces must be described precisely in a way that can be used by those who implement components with that interface and those who write programs that interact with the specified component. Mathematical notations that can be used to document these shared interfaces in a way that is both precise and readable exist now. Our methods are based on the A-7 work (e.g. [NRL84]) but have been substantively improved and tested in a variety of applications (e.g. [PLD06, SQR05, SQR06]).

Each interface specification document can be viewed as a representation of a predicate that characterizes a large set of possible implementations (variants). Only a few of these will ever be built, but a user program that is consistent with the interface as specified will be prepared to handle any of them. The specification documents may include parameters to account for visible variations in the interfaces. If such parameters are used, each implementation of the interface and each program using the interface can refer to the values of the parameters.

A set of such documents, specifying the common requirements for all the components in the product, describes the "architecture[9]" for the family of products. If the managers exercise the discipline needed to make sure that all implementations conform to these specifications, configuration-management of the resulting program family becomes relatively simple.

6. An illustrative example: car radios

is undoubtedly true, [NRL80, NRL81], show that we can plan for the future by describing the commonalities in advance.

[9] Another overused and under-defined word.

Designers and researchers who are warned that enumerating a set of known alternatives is not the right way to design a family of programs often (figuratively) throw up their hands, saying something like, “How can I possibly know all the variations that might come along in the future. They do not believe that they can be asked to state properties of some “to be developed product”. This section describes a simple example that shows how that can be done.

The example is car radios. A quick tour of some car dealers will show that they vary in a huge number of ways including the layout of the controls, the technology used for tuner and amplifier, sensors available for background noise level, the number of speakers, etc. Those who designed the simple radio in my first car would be amazed at what we can buy today. Many of the new features are implemented with software.

A manufacturer of such radios, especially one that sells a broad variety of radios to several automotive manufacturers, would like to take advantage of commonalities that exist. They must however do this without having the radios look alike or share certain features because their customers (the automobile manufacturers) want their radios to have special features that they can claim as unique selling points. The more one looks at the variety on the market and gets a sense for the rapid rate at which new capabilities are invented, the more discouraged one might become about finding a substantial set of commonalities.

However, if we focus on what capabilities **must** be there, we get a different picture. Suppose that we want to design an interface to the control panel(s). These may change from a simple set of switches and rheostats on the dash, to “soft” panels that are images on a touch sensitive display, to sequence buttons on the steering wheel to a remote control or (perhaps even a programmable controller that switches programmes on schedule). In spite of the huge differences in arrangement and function, there are certain capabilities that a radio control panel must have. Some examples are:

- It must be possible to determine if the radio should be on or off.
- It must be possible to determine if the user wants the volume raised, lowered, or kept the same.
- It must be possible to determine if the user wants to listen to the present station, to change to another pre-set station, to seek a station higher in frequency, to seek a station lower in frequency, etc.

In other words, although there are many differences, there must be common capabilities because without them the device would not be suitable as a radio control panel.

These facts allow us to identify a set of “methods” or “access programs” that we will always be able to implement, e.g. “get radio on/off switch state”. These would allow us to define a basic interface in the form of a set of programs that could be used to interact with the control panel. If the panel is replaced by another model, the interface methods might have to be implemented again, but the rest of the software need not change. The fact that implementable capabilities are usually commonalities, while the nature and format of the data can vary, is one of the reasons why an interface in the form of a set of programs is better than using an interface description language such as XML or its cousins.

However, this basic interface is not the whole story because, in addition to the superficial differences used in the illustration, there can be more substantive differences, capabilities of one unit that are not shared by all others. For example, there might be a “sleep” switch that allows the driver to request that the radio be left on for a fixed period of time before it switches off, or some “alarm” capability that increases volume at a certain time. These would not be available on all models and the basic interface would not allow them to be accessed and exploited.

This issue was also discussed in [NRL81]. We can define subfamilies with **additional** sets of interface programs. These would only be available for the sub-family. Here we see a further advantage of a program-based interface: Additional capabilities require adding interface programs but do not require modifications to existing ones. This means that existing software that used those programs need not be modified unless they want to exploit the added capability. Such code sharing is one of the ways firms can profit from applying program-family concepts.

7. Summary and conclusions

While the interest of researchers in the very practical problems associated with software product lines is welcome, it is short-sighted to focus on the problems of configuration-management, the phase in which enumeration and combinatorial analysis is possible. The greatest leverage can be obtained by focussing on the design and documentation of the family architecture at a stage where the set of variants cannot be enumerated but can be characterized. By the

time that one can enumerate alternatives, it is often too late to make a real difference.

A set of interface specifications can be interpreted as a characteristic predicate for a family of programs even though the exact membership of the family is not known. Any software product that implements the specifications correctly is a member of the family. In other words, precise documentation allows us to characterize the program family in situations where it would be impossible to enumerate the alternatives.

8. Acknowledgements

I am grateful to Marius Dragomiroiu and Ridha Khedri for several thoughtful remarks that have led to improvement of this paper.

9. References

[Hen80] Kathryn L. Heninger. "Specifying software requirements for complex systems: New techniques and their application", in *IEEE Trans. Softw. Eng.*, SE-6(1):2–13, January 1980. Reprinted as Chapter 17 in [HW01].

[HW01] Hoffman, D.M., Weiss, D.M. (eds.), *Software Fundamentals: Collected Papers by David L. Parnas*, Addison-Wesley, 2001, 664 pgs., ISBN 0-201-70369-6.

[NRL78] Heninger, K., Kallander, J., Parnas, D.L., Shore, J., "Software Requirements for the A-7E Aircraft", *NRL Report 3876*, November 1978, 523 pgs.

[NRL80] Parker, R.A., Heninger, K.L., Parnas, D.L., Shore, J.E., "Abstract Interface Specifications for the A-7E Device Interface Modules", *NRL Report 4385*, November 1980, 176 pgs.

[NRL81] Britton, K.H., Parker, R.A., Parnas, D.L., "A Procedure for Designing Abstract Interfaces for Device Interface Modules", *Proceedings of the 5th International Conference on Software Engineering*, March 1981, pp. 195-204. Reprinted as Chapter 15 in [HW01].

[NRL84] Clements, P.C., Parnas, D.L., "Experience with a Module Interface Specification Technique", *Proceedings of International Workshop on Models and Languages for Software Specification and Design*, 30 March 1984, Orlando, Florida, pp. 70-73.

[Pa76] Parnas, D.L., "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, March 1976, pp. 1-9. Reprinted as Chapter 10 in [HW01].

[Pa77] Parnas, D.L., "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems", NRL Report No. 8047, June 1977, 30 pgs. Reprinted in Infotech State of the Art Report, Structured System Development, Infotech International, 1979.

[PLD06] V. Pantelic, X. Jin, M. Lawford, and D. Parnas, "Inspection of concurrent systems: Combining tables, theorem proving and model checking", in *SERP'06: Proceedings of the International Conference on Software Engineering Research and Practice*, vol. 2, (Las Vegas, Nevada, USA), pp. 629-635, 2006

[SQR05] Baber, R., Parnas, D.L., Vilkomir, S., Harrison, P., O'Connor, T., "Disciplined Methods of Software Specifications: A Case Study", *Proceedings of the International Conference on Information Technology Coding and Computing (ITCC 2005)*, April 4-6, 2005, Las Vegas, NV, USA, IEEE Computer Society.

[SQR06] Colm Quinn, Sergiy Vilkomir, David Parnas and Srdjan Kostic. "Specification of Software Component Requirements Using the Trace Function Method", *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006)*, October 29th – November 1st, 2006, Tahiti, French Polynesia.